Microprocessors

Frame Pointers and the use of the -fomit-frame-pointer switch

Feb 25th, 2002
General Outline

Usually a function uses a frame pointer to address the local variables and parameters. It is possible in some limited circumstances to avoid the use of the frame pointer, and use the stack pointer instead.

The -fomit-frame-pointer switch of gcc triggers this switch. This set of slides describes the effect of this feature.
Consider this example

```c
int q(int a, int b) {
    int c;
    int d;
    c = a + 4;
    d = isqrt(b);
    return c + d;
}
```
The caller does something like

- push second-arg (b)
- push first-arg (a)
- call q
- add esp, 8
Stack at function entry

Stack contents (top of memory first)

- Argument b
- Argument a
- return point
- ESP
The prolog
  push ebp
  mov ebp, esp
  sub esp, 8
Stack after the prolog

Immediately after the sub of esp
second argument (b)
first argument (a)
return point
old EBP
value of c
value of d

EBP
ESP
Addressing using Frame Pointer

The local variables and arguments are addressed by using fixed offsets from the frame pointer (ESP is not referenced)

- A is at [EBP+8]
- B is at [EBP+12]
- C is at [EBP-4]
- D is at [EBP-8]
Code for q

Code after the prolog

MOV    EAX, [EBP+8]    ; A
ADD    EAX, 4
MOV    [EBP-4], EAX    ; C
PUSH   [EBP+12]        ; B
CALL   ISQRT
ADD    ESP, 4
MOV    [EBP-8], EAX    ; D
MOV    EAX, [EBP-4]    ; C
ADD    EAX, [EBP-8]    ; D
Optimizing use of ESP

We don’t really need to readjust ESP after a CALL, just so long as we do not leave junk on the stack permanently. The epilog will clean the entire frame anyway. Let’s use this to improve the code
Code with ESP optimization

Code after the prolog

MOV EAX, [EBP+8] ; A
ADD EAX, 4
MOV [EBP-4], EAX ; C
PUSH [EBP+12] ; B
CALL ISQRT
MOV [EBP-8], EAX ; D
MOV EAX, [EBP-4] ; C
ADD EAX, [EBP-8] ; D

We omitted the ADD after the CALL, not needed.
Epilog

Clean up and return

MOV ESP, EBP
POP EBP
RET

LEAVE
RET
Now we will look at the effect of omitting the frame pointer on the same example, that is we will compile this with the `-fomit-frame-pointer` switch set.

```c
int q (int a, int b) {
    int c;
    int d;
    c = a + 4;
    d = isqrt (b);
    return c + d;
}
```
Calling the function

The caller does something like

- push second-arg (b)
- push first-arg (a)
- call  q
- add  esp, 8

This is exactly the same as before, the switch affects only the called function, not the caller.
Stack at function entry

Stack contents (top of memory first)

 Argument b
    Argument a
    return point  ESP

This is the same as before
Code of q itself

The prolog
    sub   esp, 8

That’s quite different, we have saved some instructions by neither saving nor setting the frame pointer
Stack after the prolog

Immediately after the sub of esp
second argument (b)
first argument (a)
return point
value of c
value of d

ESP
Addressing using Stack Pointer

The local variables and arguments are addressed by using fixed offsets from the stack pointer

- A is at [ESP+12]
- B is at [ESP+16]
- C is at [ESP+4]
- D is at [ESP]
Code after the prolog

```
MOV   EAX, [ESP+12]   ; A
ADD   EAX, 4
MOV   [ESP+4], EAX   ; C
PUSH  [ESP+16]       ; B
CALL  ISQRT
ADD   ESP, 4
MOV   [ESP], EAX     ; D
MOV   EAX, [ESP+4]   ; C
ADD   EAX, [ESP]     ; D
```
We must remove the 8 bytes of local parameters from the stack, so that ESP is properly set for the RET instruction

```
ADD   ESP,8
RET
```
Why not always use ESP?

Problems with debugging

- Debugger relies on hopping back frames using saved frame pointers (which form a linked list of frames) to do back traces etc.
- If code causes ESP to move then there are difficulties
  - Push of parameters
  - Dynamic arrays
  - Use of alloca
Pushing Parameters

Pushing parameters modifies ESP

Sometimes no problem, as in our example here, since we undo the modification immediately after the call.

But suppose we had called FUNC(B,B)

We could not do

- PUSH [ESP+16]
  - PUSH [ESP+16]

- Since ESP is moved by the first PUSH
More on ESP handling

Once again

- PUSH [ESP+16]
  PUSH [ESP+16]

Would not work, but we can keep track of the fact that ESP has moved and do

- PUSH [ESP+16] ; Push B
  PUSH [ESP+20] ; Push B again

And that works fine
More on ESP optimization

In the case of using the frame pointer, we were able to optimize to remove the add of ESP.

Can we still do that?

Answer yes, but we have to keep track of the fact that there is an extra word on the stack, so ESP is 4 “off”.
Code with ESP optimization

Code after the prolog

MOV  EAX, [ESP+12]  ; A
ADD  EAX,4
MOV  [ESP+4], EAX  ; C
PUSH [ESP+16]  ; B
CALL ISQRT
MOV  [ESP+4], EAX  ; D
MOV  EAX, [ESP+8]  ; C
ADD  EAX, [ESP+4]  ; D

Last three references had to be modified
Epilog for Optimized code

We also have to modify the epilog in this case, since now there are 12 bytes on the stack at the exit, 8 from the local parameters, and 4 from the push we did. Epilog becomes

```
ADD ESP,12
RET
```

But no instructions were added.
Other cases of ESP moving

Dynamic arrays allocated on the local stack, whose size is not known
Explicit call to alloca

How alloca works
- Subtract given value from ESP
- Return ESP value as pointer to new area

These cases are fatal
- MUST use a frame pointer in these cases
Even better, More optimization

Let’s recall our example:

```c
int q (int a, int b) {
    int c;
    int d;
    c = a + 4;
    d = isqrt (b);
    return c + d;
}
```

We can rewrite this to avoid the use of the local parameters c and d completely, and the compiler can do the same thing.
With some optimization, we can write

```c
int q (int a, int b) {
    return isqrt (b) + a + 4;
}
```

We are not suggesting that the user have to rewrite the code this way, we want the compiler to do it automatically.
Optimizations We Used

Commutative Optimization
\[ A + B = B + A \]

Associative Optimization
\[ A + (B + C) = (A + B) + C \]

For integer operands, these optimizations are certainly valid (well see fine point on next slide)

Floating-point is another matter!
A fine Point

The transformation of
\( (A + B) + C \) to \( A + (B + C) \)
Works fine in 2’s complement integer arithmetic with no overflow, which is the code the compiler will generate.
But strictly at the C source level, \( B+C \) might overflow, so at the source level this transformation is not technically correct.
But we are really talking about compiler optimizations anyway, so this does not matter.
The optimized code

Still omitting the frame pointer, we now have the following modified code for the optimized function
The prolog

No prolog code is necessary, we can use the stack exactly as it came to us:
- second argument (b)
- first argument (a)
- return point

And address parameters off unchanged ESP

- A is at [ESP+4]
- B is at [ESP+8]
The body of the function

Code after the (empty) prolog

PUSH  [ESP+8]  ; B
CALL  ISQRT
ADD   EAX, [ESP+8]  ; A
ADD   EAX, 4

Note that the reference to A was adjusted to account for the extra 4 bytes pushed on to the stack before the call to ISQRT.
The epilog

We pushed 4 bytes extra on to the stack, so we need to pop them off

```
ADD ESP,4
RET
```

And that’s it, only 6 instructions in all. Removing the frame pointer really helped here, since it saved 3 instructions and two memory references.
Other advantages of omitting FP

If we omit the frame pointer then we have an extra register.

For the x86, going from 6 to 7 available registers can make a real difference.

Of course we have to save and restore EBP to use it freely.

But that may well be worth while in a long function, anything to keep things in registers and save memory references is a GOOD THING!
Summary

Now you know what this gcc switch does.

But more importantly, if you understand what it does, you understand all about frame pointers and addressing of data in local frames.